Optimization Principles for Collective Neighborhood Communications

Torsten Hoefler
Department of Computer Science,
ETH Zurich, Switzerland
Email: htor@inf.ethz.ch

Timo Schneider
University of Illinois at Urbana-Champaign
Urbana, IL, USA
Email: timos@illinois.edu

***Abstract*— Many scientific applications operate in a bulk-synchronous mode of iterative communication and computation steps. Even though the communication steps happen at the same logical time, important patterns such as stencil computations cannot be expressed as collective communications in MPI. We demonstrate how neighborhood collective operations allow to specify arbitrary collective communication relations during runtime and enable optimizations similar to traditional collective calls. We show a number of optimization opportunities and algorithms for different communication scenarios. We also show how users can assert constraints that provide additional optimization opportunities in a portable way. We demonstrate the utility of all described optimizations in a highly optimized implementation of neighborhood collective operations. Our communication and protocol optimizations result in a performance improvement of up to a factor of two for small stencil communications. We found that, for some patterns, our optimization heuristics automatically generate communication schedules that are comparable to hand-tuned collectives. With those optimizations in place, we are able to accelerate arbitrary collective communication patterns, such as regular and irregular stencils with optimization methods for collective communications. We expect that our methods will influence the design of future MPI libraries and provide a significant performance benefit on large-scale systems.**

I. Introduction

The Bulk Synchronous Parallel (BSP) style of programming using the Message Passing Interface (MPI) is one of the most popular parallel programming paradigms in High Performance Computing. BSP applications exhibit characteristic communication and computation phases, called supersteps [1]. The inherently clear division of functionality supports modularity (abstraction of communication layers) and ease of programming (all processes are in a similar global state that is easy to reason about). Some static communication patterns fit one of the pre-defined collective communication operations in MPI, for example, the global alltoall of a Fast Fourier Transform, or the allreduce convergence check of many Monte Carlo methods.

Traditional collective communications always include some form of global synchronization where either all processes synchronize with one process (the root) or all processes synchronize with all other processes. Also, communication typically scales with $\Theta(log(P))$ time and a total number of $\Theta(P \cdot log(P))$ messages on P processes. Newer parallel algorithms try to reduce such global dependencies by only using limited or sparse communications where each process communicates only with a small subset of the whole process set.

This inherently more scalable programming technique is unfortunately not supported well by traditional collective operations and users often resort to point-to-point communication to implement highly scalable algorithms. However, a collective formulation of a communication superstep may not only improve performance and performance portability significantly [2], but it can also improve readability and maintainability [3] of application code.

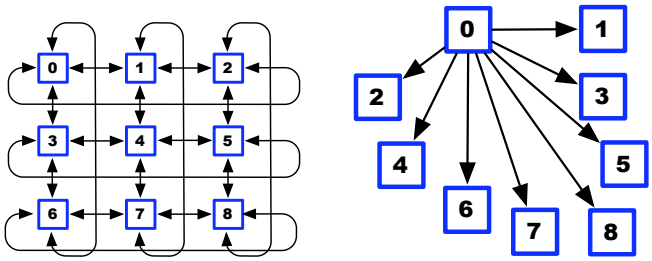
MPI defines only a set of 17 common communication patterns as collective communication functions [4, §5]. Important sparse communication patterns such as 2d and 3d Cartesian neighborhoods that occur in many stencil computations, or irregular process patterns, that may result from load balancing or adaptive mesh refinement, are not supported. In order to close this gap, the MPI Forum added "neighborhood collective operations", two new operations (with three variants each). Those operations allow the user to specify *arbitrary* communication topologies as collective operations and provide additional optimization opportunities to the MPI library.

The MPI-3.0 standard will include neighborhood collectives as part of the process topology chapter. Process topologies can either be constructed as n-dimensional Cartesian topologies (with optional wrap-around) or arbitrary (distributed) graph topologies [5][1]. Neighborhood collectives can be used to communicate along the edges of the resulting graph. Thus, neighborhood collectives either communicate along an n-dimensional regular process grid on Cartesian communicators or in arbitrary user-defined neighborhoods on general or distributed graph communicators. Therefore, a user can express any collective communication pattern, including all predefined collective communications, as neighborhood collectives.

Figure 1 shows two example graphs: (a) a Cartesian graph with nine processes in a 3x3 grid, which may result from a 5-point stencil computation, and (b) a graph communicator with nine processes that reflects the logical communication pattern

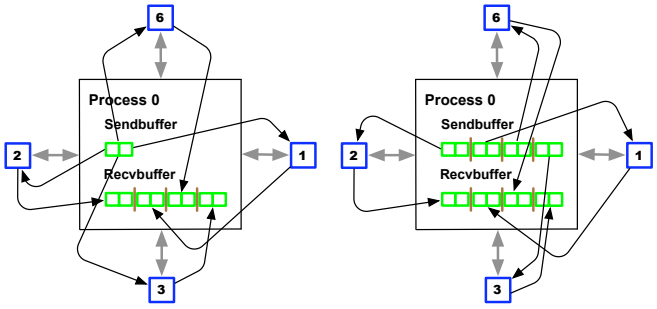
SC12, November 10-16, 2012, Salt Lake City, Utah, USA
978-1-4673-0806-9/12/$31.00 2012 IEEE

[1]Using the MPI-1.0 graph topology is possible but not recommended due to its scalability issues.



(a) 2D Cartesian (b) Scatter/Bcast

Fig. 1. Example Process Topologies



(a) Allgather (b) Alltoall

Fig. 2. Neighborhood Allgather vs. Alltoall

of a broadcast or scatter communication.

Neighborhood collectives can be seen as generalized collective communications. In this work, we show how to apply techniques known from optimizing traditional collective communications toward generalized neighborhood collectives. We also show how additional information about the application code can lead to portable optimizations that enable performance portable execution across different architectures.

We focus on MPI in our discussions, mainly due to its wide availability and use, however, we point out that the concept of neighborhood collectives is general in parallel programming and all developed optimization techniques can be applied in different environments. In fact, our software stack is divorced from MPI and allows for easy application to other relevant parallel programming frameworks or runtime libraries. The following section discusses the MPI-3.0 interface.

A. Neighborhood Collectives in MPI-3.0

The concept of neighborhood collectives has initially been motivated under the name *sparse collectives* in [6], [7]. The MPI Forum simplified and renamed the proposed functions. The two main functions are MPI_Neighbor_allgather and MPI_Neighbor_alltoall. As described before, the communication pattern for each function is specified by an MPI process topology. The main difference between the functions are the local buffer semantics: while allgather is sending the same buffer to all destinations and receives into distinct buffers from all sources, alltoall sends different (personalized) buffers to all destinations while receiving into different buffers from all sources. Thus, the example graph shown in Figure 1(b) would be equivalent to an MPI_Bcast when using MPI_Neighbor_allgather and it would be equivalent to MPI_Scatter when using MPI_Neighbor_alltoall on the shown topology. Figure 2 illustrates the semantic difference in buffer access for the 2D Cartesian graph of Figure 1(a). Bold gray arrows are the neighborhood relations specified in the graph and thin black arrows show the data movement from send and to receive buffers at process 0.

The following code fragment shows the simple construction of the Cartesian communicator and an allgather communication where the integer value in `sbuf[0]` is sent to all four neighbors and the integer sent by neighbor `i` (in dimension order) is received in `rbuf[i]`.

```
int dims[2]={2,3}, periods[2]={1,1};
MPI_Comm comm_cart;
MPI_Cart_create(MPI_COMM_WORLD, 2, dims, periods, 0,
    &comm_cart);
int sbuf[1], rbuf[4];
MPI_Neighbor_allgather(sbuf, 1, MPI_INT, rbuf, 1,
    MPI_INT, comm_cart);
```

A call to alltoall looks fairly similar but specifies a send buffer with four elements and sends the value in `sbuf[i]` to neighbor `i`. MPI also specifies vector variants (e.g., MPI_Neighbor_allgatherv) and variants with neighbor-specific datatypes (e.g., MPI_Neighbor_allgatherw) that follow the usual MPI semantics. All neighborhood collectives have a blocking and a nonblocking interface.

The distributed graph topology interface (example omitted for brevity) accepts an additional MPI_Info argument where the user can communicate additional optimization hints or assertions about his code to the MPI library. The standard suggests to use info arguments to communicate semantics of edge weights for topology mapping, but we identify other optimization possibilities in the following.

During the neighborhood creation call (either the Cartesian or a graph constructor in MPI), the communication library can optimize the communication topology, similarly to the optimizations that are performed during communicator creation (e.g., algorithm selection [8], [2]). Yet, unlike traditional collective operations, different edges in the topology may carry different communication volumes which makes optimizations harder. An info argument could be used to assert that all edges carry the same load in order to re-gain this knowledge. However, we also demonstrate the applicability of our scheme in the general case of varying edge loads.

In this work we show how neighborhood collectives together with info arguments enable the user to specify detailed information about the memory access and communication pattern of a collective communication step. This enables a plethora of communication opportunities that we describe and classify in the following.

The key contributions of our work are:

1) We classify different levels of persistence (static properties) of collective communications in parallel codes and propose a mechanism to specify this knowledge in a portable way to the runtime library.
2) We develop an open-source execution and optimization framework for arbitrary nonblocking neighborhood collectives.
3) We propose and evaluate different optimization techniques for arbitrary collective operations to improve performance.
4) We demonstrate the applicability of our approach with a wide set benchmarks and applications.

II. Persistence Levels in Collective Communication

Numerous optimizations in different communication models (cf. [9], [10], [11] among many others) have been proposed to optimize predefined collective communications with known static patterns. Common techniques are to schedule communications in order to avoid congestion and to use message coalescing and forwarding through proxy processes to reduce injection rate limitations.

We now describe how those techniques can be used dynamically, i.e., during runtime, to optimize arbitrary neighborhood communications. As with most optimizations being able to make additional assumptions about the problem that constrain the optimization space leads to higher optimization potentials. For neighborhood collectives, such assumptions can be made about the communication topology, sizes, and buffer access. We define three "persistence levels" that indicate how fixed each of the parameters is across iterations.

The user can communicate persistence properties of his application through passing special info keys that guide the optimization of neighborhood collectives. This scheme is completely transparent and portable within the MPI standard. If an implementation does not understand an info key, then the key is ignored and the code executes as if the key was not specified. Thus, info keys can be used to specify hints about the code but they cannot change the structure of the communication. We point out that users are not allowed to "lie" with specified info keys, i.e., if a user specifies a certain code property, the code must act within the specified limits. Also, info keys for MPI topologies are collective, i.e., all processes have to specify the same set of keys for the topology construction.

If an application has multiple repeating communication steps with different sizes or collectives, the user can create one topology communicator for each call-site and provide the highest optimization options. Creating a communicator for each set of arguments may seem expensive, however, we point out that communicators that only differ in their info arguments can be stored efficiently with simple lossless compression techniques with a constant ($\mathcal{O}(1)$) time and space overhead.

Our optimizations target networks that are able to perform remote direct memory accesses (RDMA). This type of network recently gained such high popularity due to it's simple hardware implementation and high performance, that is is available on all current large-scale supercomputers. This section describes high-level approaches to optimize for RDMA networks while later experiments show the benefits of one particular implementation.

We now discuss the different persistence levels for neighborhood collectives.

A. Persistent Communication Topology

By definition, the process topology specifies communication relations for neighborhood collectives persistently. MPI process topologies are immutable, i.e., in order to change the communication topology, the user has to create a new topology object.

Knowledge about the exact communication topology enables the implementation to statically allocate resources for the communication, perform communication scheduling and build efficient structures for communication protocols. Since topology communicators still allow arbitrary communications, the user may set the info key "cdag_strict_communication"[2] to "true" to communicate that communication is only performed along edges of the topology.

1) Fixed Communication Channels: The library can establish fixed communication channels between each pair of connected processes and it can initiate communication protocols. Traditional MPI communications need to support P^2 connections among P processes with P eager buffers per process [12]. Setting "cdag_strict_communication" to "true" may result in a significant memory overhead reduction if the implementation utilizes eager messaging with fixed buffers per communication partner.

2) Synchronization Trees: The library could also build tree structures as it would do for standard collectives, however, since data-sizes are not known, the best shape of the tree cannot be determined. Nevertheless, such trees can be built for notification purposes to support collective synchronization or rendezvous messaging (cf. Section II-C2). Having a static communication topology also allows to determine good message schedules to avoid endpoint congestion or utilize different communication mediums.

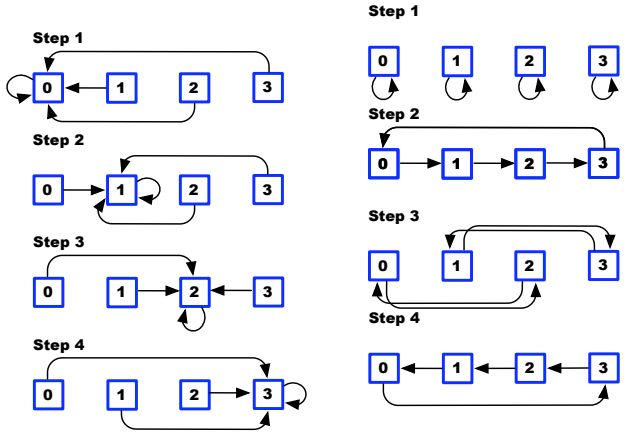
3) Communication Scheduling: The implementation can also decide in which order messages are scheduled to the different destinations. This could either be influenced by the transport medium to the destination (e.g., shared memory or off-node network) or by the load at the source or the destination (to avoid hot-spot traffic).

In order to maximize communication/communication overlap, we build a static schedule that issues all off-node communications using an RDMA offloading mechanism and then performs shared memory copies which occupy the CPU (cf. Section III-B2). Ideally, both communications overlap completely.

We apply a second optimization to reduce endpoint congestion. For this, we color the edges of the communication graph

[2]We chose the "cdag" (communication DAG) prefix to all our info keys to avoid namespace pollution.

using a greedy coloring heuristic developed by Welsh and Powell [13]. The colors then correspond to the communication rounds. This idea was first discussed by Hoefler and Träff [6] but not tested in practice. Its utility is demonstrated in the following example of an alltoall communication generated with the code `for(i=0; i<p; ++i) send(i)` on all processes $0 \leq i < p$. In the first step, all processes send to process 0, in the second step to process 1, ..., in the i-th step to process i. This results in a total bandwidth of $p \cdot \frac{bw}{p}$ if each process contributes its local injection bandwidth bw. One possible (ideal) coloring would be represented by the loop `for(i=0; i<p; ++i) send((r+i)%p)` assuming r is the calling processes' rank. This coloring would lead to a total bandwidth of $p \cdot bw$. Figure 3 shows the original schedule and one possible ideal coloring.



(a) Simple Algorithm (b) Ideal Coloring

Fig. 3. Alltoall Example illustrating Schedule Coloring

B. Persistent Communication Topology and Message Sizes

If the message sizes are known at topology creation time, then the user can provide a weighted topology graph to the MPI library and indicate that the specified weights represent the exact communication sizes by setting the info key "cdag_exact_byte_communications" to "true".

Knowing the absolute message sizes statically allows to perform intelligent re-ordering and forwarding through intermediate processes.

1) Tree Transformations: We now show a scalable distributed algorithm to determine a good communication schedule based on the parameters of the communication network and the message sizes. For this, we consider a simplified version of the LogGP model [14]. In LogGP, one assumes a communication operation among P processes, a maximum network latency of L, a per-message overhead of o, an injection overhead (at the network level) of g, and a time per byte sent of G. To simplify the model, we assume $o > g$, which renders g insignificant. We assume a globally fixed uniform message size of k Bytes.

Since each message requires a constant overhead o to be sent, a possible optimization would be to balance the number of messages from high-outdegree processes to low-outdegree processes. This can be repeated for multiple iterations to create multiple tree levels. The algorithm stops when a convergence criteria is met. The main gain of the tree balancing is the distribution of o across multiple processes for neighbor alltoall and the better bandwidth utilization for neighbor allgather, respectively.

a) Neighbor Alltoall: The tree optimization algorithm for alltoall repeats the following two steps until it converged:

Step 1: Determine process p_m with most outgoing edges m and process p_s with the smallest number of outgoing edges s.

Step 2: If $m - s > t$ move $\frac{m+s}{2} - s - \left\lceil \frac{L}{o} \right\rceil$ messages from p_m to p_s else stop iterating.

The subtraction of $\left\lceil \frac{L}{o} \right\rceil$ in the second step exists because process p_m balances its excess load to p_s which in turn can only start sending after it received the message (L) and process p_m can continue sending $\left\lceil \frac{L}{o} \right\rceil$ messages during that time. As with the usual scatter tree optimizations, this *only applies for small messages (small k)* when the additional bandwidth consumed by message forwarding is not the bottleneck. Figure 4 shows an example where a hot-spot neighbor alltoall pattern is transformed to a scatter tree which is similar to an optimal scatter tree in LogGP [14].

b) Neighbor Allgather: The tree optimization algorithm for allgather repeats the following two steps until it converged:

Step 1: Determine process p_m with most outgoing edges m and process p_s with the smallest number of outgoing edges s.

Step 2: If $m - s > t$ move $\frac{m-s}{2} - \left\lceil \frac{L}{o+(k-1)G} \right\rceil$ messages from p_m to p_s else stop iterating.

The subtraction of $\left\lceil \frac{L}{o+(k-1)G} \right\rceil$ in the second step exists because process p_m balances its excess load to p_s which in turn can only start sending after it received the message (L) and process p_m can continue sending $\left\lceil \frac{L}{o+(k-1)G} \right\rceil$ messages during that time. This algorithm can be used for arbitrarily large k because it does not increase the total communication volume if forwarding processes are neighbors of the source. The example shown in Figure 4 is also illustrating a valid allgather optimization (messages 1, 3, and 5 are identical in this case).

The parameter t is generally architecture-dependent. We chose $t = 2$ for all our experiments. Both heuristics converge rather quickly in practice, e.g., for the hotspot pattern in $\mathcal{O}(\log P)$ steps and each step has a cost of $\mathcal{O}(\log P)$ (allreduce). Thus, our balancing algorithm is scalable to large process counts. However, the resulting tree may not be optimal in the LogGP model. Optimal tree configurations (cf. [14]) are an interesting direction for future work.

C. Persistent Communication Buffers

If not only the message sizes but also the communication buffers are static, i.e., the buffer addresses and sizes of **all**

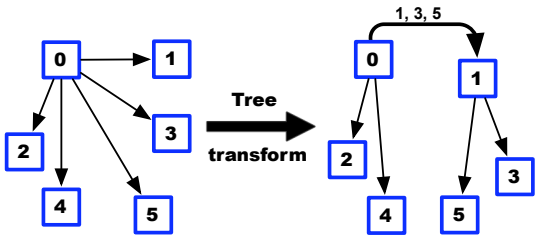


Fig. 4. Tree Transformation Example

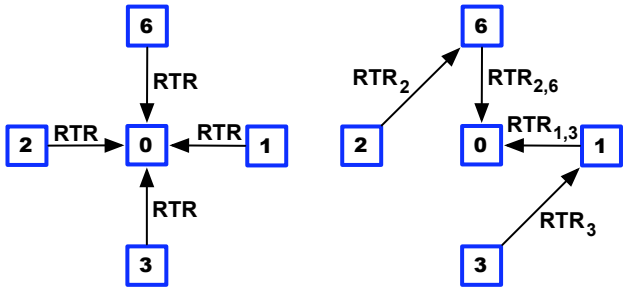
calls to the same neighborhood collective on this communicator are the same, then the user may set the info argument "cdag_static_buffers" to "true". Having static buffer addresses allows similar optimizations as for persistent point-to-point messages, however, those optimizations can now be applied in a collective manner, taking advantage of the knowledge about the full communication schedule.

1) Static RDMA Regions: Static buffers allow to register the buffer memory to the network interface at the first call. The keys to access (RDMA remote access handles) are then exchanged along all communication edges and all the information is saved for future calls to the same function. The next call will then simply "fire off" the communications which have all RDMA descriptors already prepared. The direct memory access to remote nodes enables also advanced synchronization protocols.

Having static information about the communication buffers enables us to apply static protocol optimizations. Every RDMA communication requires two synchronizations: (1) Ready to Receive (RTR), where the receiver needs to indicate that the buffer is ready for remote writes (i.e., the receiver entered the collective communication function) and (2) Ready to Exit (RTE), where sender notifies the receiver that the data communication is finished and the receive buffer is in a consistent state. Figure 5(a) illustrates the state of the art point-to-point RDMA synchronization scheme [12]. A collective communication function can return when a process received RTEs from all neighbors and finished its own sends.

2) Collective RDMA RTR Protocol: The RTR and RTE protocols can be optimized separately. The RTR protocol can be performed collectively, e.g., a global barrier may be used at the beginning of the collective call to communicate RTR. This may be a good strategy if the machine offers fast ($\mathcal{O}(1)$ time) global barrier support, like BG/L and BG/P [15]. However, if the barrier is implemented with point-to-point messages, this will add $\Omega(\log(P))$ communication overhead.

Instead of a global barrier, we propose a collective RTR protocol on each process' neighborhood using neighborhood RTR trees with each sender as root. This protocol *combines* RTR messages in a tree shape and can reduce the RTR time from $\Omega(\tau)$ to $\mathcal{O}(\log \tau)$ for a process with τ neighbors. The desired degree of the tree depends on the system parameters (L/o as discussed before). Figure 5(b) shows a direct comparison between the traditional model (with four RTR messages going to the root) and the optimized tree model (with two RTR messages going to the root) for process 0 in the 2D Cartesian topology in Figure 1(a). We remark here that most of today's networks require trees with an outdegree larger than two.



(a) Traditional point-to-point protocol

(b) Collective tree protocol

Fig. 5. Illustration of our collective tree RDMA protocol. RTE was omitted from this figure for readability because it would simply invert the RTR arrows.

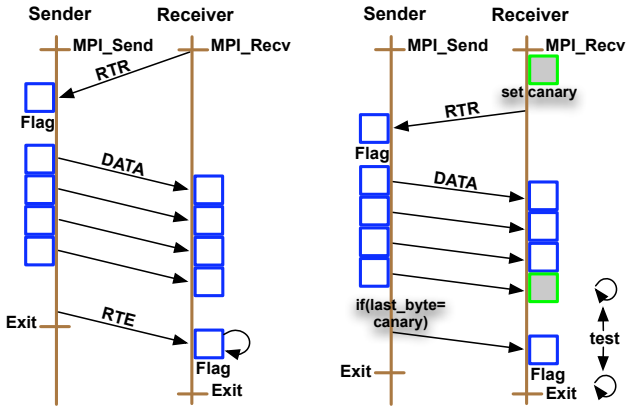
3) Canary RTE Protocol: RTE can be optimized with a similar collective protocol like RTR. However, for small messages, we propose a protocol which, in the common case, works without additional communications. This protocol relies on the ordering semantics of the network and memory buses. If one can ensure that all bytes are committed in order (or at least the last byte is committed last), one can use the last byte (or some bytes) as *canary value* to check for communication completion. For this, we pick a constant C as canary. When the collective function is entered, we initialize all last bytes of the receive buffers of each neighborhood communication to C before we start the RTR protocol. RTE is detected as soon as the canary value changes from C. However, this protocol would not terminate if the last byte of the transmitted user-data is identical to C. For this case, we are using an additional flag outside the user data which may also indicate completion. If the sender detects that the last byte in the send buffer is equal to C, then it triggers the additional flag at the receiver (with an additional RDMA put). The receiver then completes if either the canary value changes from C or the additional flag is triggered. This protocol will save the additional message in most of the cases. The likelihood of conflict can even be reduced if multiple bytes are used to encode the canary value C. Figure 6 shows a comparison of the traditional RDMA protocol (Figure 6(a)) and our optimized canary protocol (Figure 6(b)).

III. Implementing Nonblocking Neighborhood Collectives

We now discuss how we use the optimization principles described in the previous section in order to implement high-performance non-blocking nearest neighborhood collective operations in our open-source framework.

A. Reference Implementation

The reference implementation for MPI-3.0 standardization of neighborhood collectives in the MPI Forum is integrated in



(a) Traditional RDMA Protocol

(b) RDMA Protocol with Canary Value

Fig. 6. Illustration of the Canary RDMA Protocol.

LibNBC [16]. LibNBC uses an internal schedule to represent multiple stages of nonblocking collectives [16]. However, the specific implementation of neighborhood collective operations simply starts all corresponding nonblocking MPI sends and receives and thus only utilizes a single stage.

B. Remote Direct Memory Access

We use two different low-level RDMA transport mechanisms for our implementation: DMAPP [17] for off-node communication over Cray's XE6/XK6 Gemini interconnect [18], which is directly connected to AMD's Hypertransport [19], and XPMEM [20] Cray's on-node shared address space communication layer. Both layers allow direct remote memory access.

1) DMAPP: DMAPP supports two request modes: Fast Memory Access (FMA) and Block Transfer Engine (BTE). In the FMA mode, the CPU is used to push the data to the network interface (NIC), which is fastest for small messages. The BTE mode programs the DMA controller, which has some constant overhead but then enables asynchronous message transmission. We will describe how we use both protocols in our implementation.

DMAPP also allows the user to select deterministic in-order, node-hash, or adaptive routing. We can only use the canary RTE protocol up to the Gemini packet size of 64 Bytes because Gemini (AMD Hypertransport) does not guarantee that messages are committed in-order to memory. We always use the adaptive routing mode because message striping results in a higher bandwidth. We generally use non-blocking communication with bulk completion semantics for DMAPP.

2) XPMEM: XPMEM is a kernel module that enables process A to directly map and access process B's memory space if process A and B are running within the same coherent operating system image. We use XPMEM to allow direct remote access to the send and receive buffers just as RDMA would do for off-node. The main difference is that the data is always copied by the CPU (we utilize standard libc memcpy).

C. Dependency Graph Communication

We use a general directed acyclic graph (DAG) formulation to specify the local communication schedule on each process. When a process topology is specified, the DAG represents all neighborhood relations and is distributed so that each process knows its immediate neighbors (destinations and sources) locally.

The DAG allows to express dependencies between two operations—A and B which mean that operation B can only be executed after operation A finished. Operations can be send, recv, or local copy operations. This functionality is needed to implement the message scheduling and tree reordering optimizations.

1) Applying Tree Optimizations: Depending on the persistence level (topology, sizes, or buffers), we perform static optimizations described in Section II either during the graph creation (for topology and tree reordering) or during the first call (binding buffers) to the collective operation. Those optimizations are performed by transforming the DAG. The coloring heuristic introduces dependencies between the send vertices which forces them to be executed in the order of their color. The tree reordering, which moves communication to a proxy process, may add local copy vertices to pack non-consecutive buffers, and moves send vertices (including the remote memory identifiers) to the remote process.

Each optimized DAG is then cached at the communicator and re-used to execute calls to the respective collective operation.

2) Communication DAG Execution: The basic idea was presented in [21], we now discuss how we use dynamic communication scheduling to implement optimized nonblocking neighborhood collective operations over an RDMA network. Each send and recv call is encoded as a vertex in a dependency

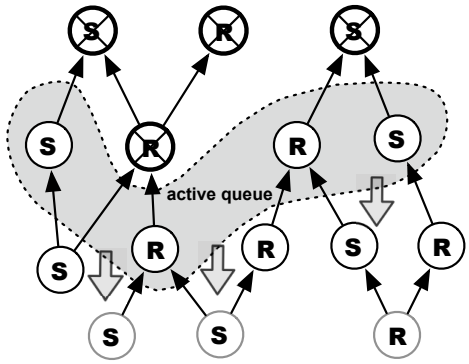


Fig. 7. DAG Scheduling

DAG. The scheduler starts with the DAG and puts all roots (vertices without dependencies) into the active queue (AQ). Then the scheduler iterates over the queue and progresses each operation (vertex) in the active queue and checks it for completion. Upon completion of a vertex, the scheduler removes all dependencies to this vertex and starts all vertices with no remaining dependencies. Using this scheme, the scheduler is

essentially moving in a wave-front through the whole DAG schedule. This is illustrated in Figure 7 where the crossed operations already finished executing and the operations in the wavefront (active queue) are being executed.

Each of the vertices can be represented by a state machine which expresses the progression through the RDMA communication protocol. For brevity, we only demonstrate the state machine for an DMAPP small message receive vertex. The first state is the initial state after it was added to AQ. Right after adding it, the receiving process initializes the canary value to C and sends an RTS message to notify the sender. It then moves to state 1 where it remains in a polling loop until either the canary value changes or the notification bit is toggled. After any of those events, the receive finishes. Figure 8 shows the state state machine for a DMAPP small message receive. All other operations utilize similar state machines to implement lower-level protocols.

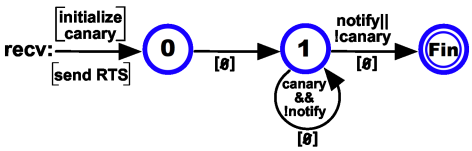


Fig. 8. DMAPP Small Message Send Protocol State Machine

IV. Experimental Evaluation

To evaluate each optimization technique, we analyzed microbenchmark performance with manually crafted communication patterns and various message sizes to show the impact of each optimization in isolation. We then extracted several communication patterns from real-world applications and examined the resulting microbenchmarks. In addition, we modified one application kernel and a full application to support neighborhood collective communications.

Our implementation supports nonblocking and blocking variants of all collectives. However, all experiments were done with the blocking implementation of neighborhood collectives because transforming applications to use nonblocking collectives is a more complex task than simply introducing neighborhood collectives. Evaluating tradeoffs and benefits for nonblocking neighborhood communication is an interesting future work.

A. Experimental Environment

For our experiments, we use the Blue Waters Test System (JYC), a single cabinet Cray XE6 (approx. 50 nodes with 1600 Interlagos 2.3-2.6 GHz cores). We use the GNU compiler version 4.6.2 in the Cray compiler environment version 4.0.46.

B. Relevant Communication Patterns

In this section, we will demonstrate each key optimization with representative communication patterns. We start with a sparse alltoall pattern to demonstrate the benefit of message scheduling (coloring) and then show different real-world Cartesian stencil examples.

None of those sparse communication patterns can be represented by MPI's previous collective functions. Thus, we use the most common way to implement such exchanges: post all receives nonblocking (MPI_Irecv), start all sends nonblocking (MPI_Isend) and wait for all sends and recvs to complete (MPI_Waitall). We confirmed that this is the fastest method to implement such exchanges on our test system.

1) Sparse Alltoall Pattern: A sparse alltoall pattern $\mathcal{A}(s,p)$ is specified by a data size s and a parameter p. The parameter p ($0 \leq p \leq 1$) indicates the probability that process i sends a message of size s to process j (i.e., there is an edge from i to j in the neighborhood collective), independent from any other process pair. The resulting graphs are essentially random Erdős-Rényi graphs [22].

Figure 9 shows the alltoall pattern with varying density on 1024 processes and communicating 16 bytes along each edge. Coloring was used to improve the ordering of messages. The

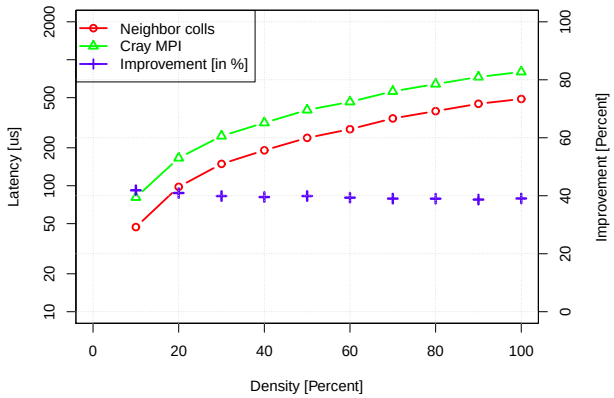


Fig. 9. Alltoall performance with varying density

lines show the absolute latency for neighborhood collective and Cray MPI and the blue crosses show the relative improvement of neighborhood collectives in percent. We observe a consistent performance advantage about 40% across all densities.

In the following, we create representative application communication patterns that are widely used in scientific applications. We assume that communication buffers are persistent, which is true for all investigated applications.

2) Cartesian Stencil Communication: Regular stencils are used in many statically decomposed PDE and ODE solvers. Common stencils are either two-dimensional (e.g., [23]), three-dimensional (e.g., [24]), or four-dimensional (e.g., [25]). Stencils communicate along each dimension in two directions, creating a total of $2d$ communication edges in a d-dimensional stencil (assuming periodic boundary conditions). Figure 1(a) shows a two-dimensional stencil.

We create an d-dimensional stencil with P processes by using MPI_Dims_create to get the best decomposition and MPI_Cart_create to create a Cartesian topology.

Figure 10 compares the performance in an alltoall over a 2d Cartesian topology (each process has four neighbors as results from a 5-point stencil) on 512 processes. This pattern

is similar to the communication in the Weather Research and Forecast Code (WRF).

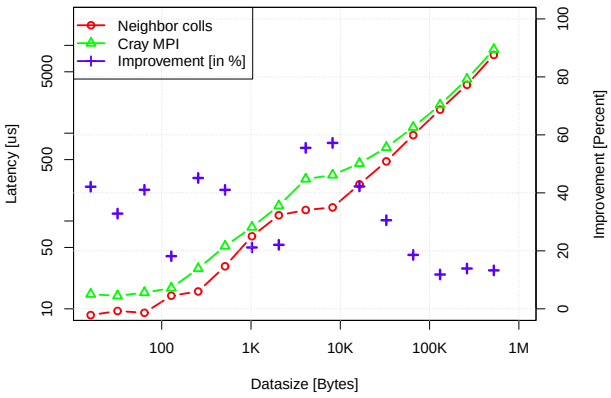


Fig. 10. 2d Cartesian Performance

We benchmark different data sizes from 16 Bytes to 1 MiB. The lines represent our neighborhood collectives versus Cray MPI's performance and the crosses show the improvement in percent (right scale). Small messages show the largest improvement because of all the reduced overheads (static matching, canary protocol, etc.) down to 50% of the original communication time, a speedup of 2x. Larger messages show less improvement (≈15%) due to the bandwidth boundedness for both communication schemes.

Figure 11 compares the performance of a neighbor alltoall over a 4d Cartesian topology (each process has eight neighbors as results from a 9-point stencil) on 512 processes. This pattern is similar to the communication in MIMD Lattice Computation (MILC). We again observe a high benefit for small messages,

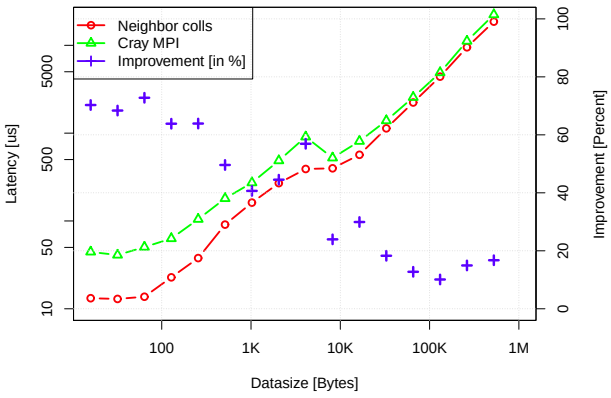


Fig. 11. 4d Cartesian Performance

around 75%, a speedup of 4x. Again, the speedup falls to about 18% for large messages due to the bandwidth-boundedness of the problem. We observe protocol changes in MPI (at 4 kiB) and DMAPP (at 8kiB) in both graphs.

C. Real World Applications

We also equipped real-world applications with neighborhood collectives. Changing MPI BSP-style applications to use neighborhood collectives is often simple because the structure is very similar to existing codes and may even lead to significant code simplifications, cf. [7]. For brevity, we show two representative cases for two classes of applications working with regular and irregular grids. The first case is the Weather Research and Forecast Model which already implements an MPI Cartesian communicator on a regular 2D grid. The second example is a matrix-vector multiplication kernel which is representative of many irregular applications.

1) The Weather Research and Forecast Model: The Weather Research and Forecasting (WRF) Model [23] is a mesoscale numerical weather prediction system designed to serve both operational forecasting and atmospheric research needs. We used the WRF-ARW core version 3.3.1 which is based on an Eulerian solver for fully compressible nonhydrostatic equations, cast in flux (conservative) form, using a mass (hydrostatic pressure) vertical coordinate. The solver uses a third-order Runge-Kutta time-integration scheme coupled with a split-explicit 2nd-order time integration scheme for the acoustic and gravity-wave modes.

We used the "em_b_wave" input set with five simulation days (720 iterations with 600 seconds each) using 215,865 total grid points. We ran this set in strong scaling mode on 32-512 cores using 32 OpenMP threads per process. We purposely chose this small system to emulate a strong scaled run on many processors.

Figure 12 shows the improvement of the communication phase of the WRF code with neighborhood collectives versus the original implementation. We observe a 40% maximum

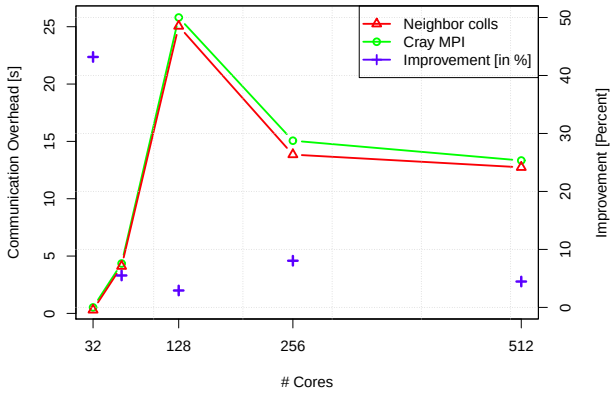


Fig. 12. WRF Communication Performance

improvement which translates to 15% application speedup and an average of 7-10% improvement which translates to 3-5% application speedup.

2) Sparse Matrix-Vector Multiplication: The sparse matrix vector multiplication represents a large number of irregular codes. For this example, we load a matrix from the UFL Sparse Matrix collection [26] to ensure a realistic input. We then partition this matrix using ParMETIS [27] and perform a parallel sparse-matrix vector multiplication with an 8-byte allreduce at each iteration. This represents a typical conjugate

gradient iteration as used in many parallel applications, e.g., MILC [25] or Algebraic Multigrid [28].

We use the matrices "aug2dc", "andrews", and "tube2" and compare Cray MPI's performance to optimized neighborhood collectives. Figure 13 shows the relative performance if neighborhood collectives (lower is better). Neighborhood

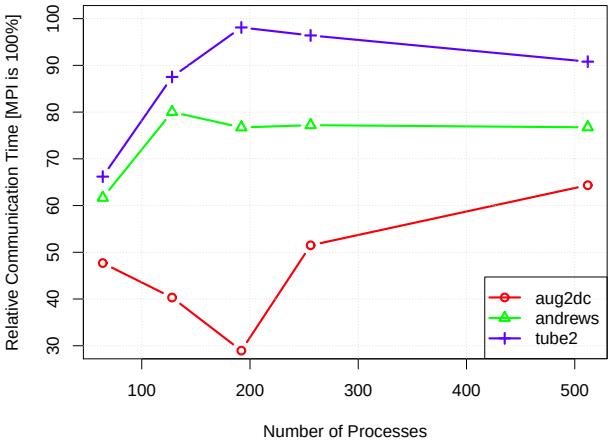


Fig. 13. Matrix Vector Communication

collectives show a reduction in communication time between 5% and 70%. The curves are not smooth because this is a strong scaling run and ParMETIS partitions the matrices with different edge cuts, depending on the size of the allocation.

V. Related Work

Neighborhood collective operations have been introduced by Hoefler and Träff in [6] and later been proposed for standardization to the MPI Forum. Even though the concept of neighborhood or sparse collective operation itself is new, we were able to borrow several algorithmic principles to drive our heuristics from established collective communication primitives such as efficient alltoall implementations [29] or scatter/gather optimizations [14].

Vetter and Mueller show that scalable applications most often depend on point-to-point communications where each process communicates with 3-7 distinct processes (neighbors) [30]. Kamil et al. conclude in a similar study that other relevant highly scalable applications applications communicate small messages (less than 2 kiB) with up to 66 neighbors [31].

Several related works discuss optimizations of stencil computations with regards to I/O (out-of-core methods) [32] or communication-minimizing tiling strategies [33], [34]. Our work focuses on the optimization of the communication step itself and is thus complementary to this rich body of stencil optimizations.

Potluri et al. [35] demonstrated MPI One Sided techniques to optimize different communication patterns. Gabriel et al. [36] optimize stencil communications by adaptively choosing different MPI-based point-to-point communication protocols. Both works experimented with different communication options without utilizing the collective nature of the communication in the optimization techniques.

Kumar et al. [37] use BlueGene's specialized hardware support to optimize neighborhood collective operations. However, their proposed interface cannot work with the MPI-3.0 standard interface and makes several simplifying assumptions.

Saltz et al. use the inspector/executor principle to exploit locality in the communication pattern of irregular applications. An inspector monitors the communication pattern and an executor optimizes the corresponding communication call. Das, Saltz et al. [38] list several optimizations, such as double send elimination and message coalescing for this scenario. However, all of their optimizations only target point-to-point communications and do not perform global schedule optimizations (coloring or proxy sends) like our approach.

Other related works such as STAR-MPI [39] and ATCC [8] select different collective operations based on varying input parameters and environment changes. However, those schemes simply select from a set of different implementations of predefined collective operations.

VI. Conclusions and Future Work

In this work, we showed the usefulness of nonblocking neighborhood collective operations and demonstrated several optimization approaches. We defined three different persistence levels, topology, data sizes, and buffer addresses that enable a hierarchy of optimization options. We describe each optimization in detail and show performance results comparing different schemes. The two key optimizations, message coloring and tree reordering are derived from traditional optimized collectives and show significant performance benefits on a Cray XE6 system.

Our high-performance implementation of neighborhood collectives is up to a factor of four faster than the highly optimized Cray MPI for small point-to-point messages in a 9 point (4d) stencil communication. We also achieve a 15% higher bandwidth for large stencil messages. We show application communication speedups as high as 15%.

As future work, we plan to employ runtime-autotuning to tune the large number of parameters (e.g., tree degree, protocol choices etc.). We also plan to offload our optimization engine to a separate core so that it can constantly optimize communication schedules while the application is running. Enabling the new schedules is then just a matter of broadcasting the new schedule to all processes and swapping a pointer. We also plan to experiment with optimizations for the underlying network topology.

We expect that our work will act as a template for the optimized implementation of neighborhood collectives. The importance of (localized) neighborhood collectives is growing with the system size, so we expect that the described principle of optimizing communications in their neighborhoods will become a generic optimization principle for large-scale applications.

Acknowledgments

We thank Larry Kaplan, Duncan Roweth, Howard Pritchard, and Mark Pagel from Cray Inc. for support on how to use

DMAPP and XPMEM efficiently. We thank Mark Straka and Robert Gerstenberger for support with WRF. This work was partially supported by the DOE Office of Science, Advanced Scientific Computing Research, under award number DE-FC02-10ER26011, program manager Lucille Nowell and by the Blue Waters sustained-petascale computing project, which is supported by the National Science Foundation (award number OCI 07-25070) and the state of Illinois.

REFERENCES

[1] L. G. Valiant, "A bridging model for parallel computation," *Commun. ACM*, vol. 33, no. 8, pp. 103–111, 1990.

[2] R. Thakur, "Improving the performance of collective operations in mpich," in *Recent Advances in Parallel Virtual Machine and Message Passing Interface. Number 2840 in LNCS, Springer Verlag (2003) 257267 10th European PVM/MPI Users Group Meeting*, pp. 257–267, Springer Verlag, 2003.

[3] S. Gorlatch, "Send-receive considered harmful: Myths and realities of message passing," *ACM Trans. Program. Lang. Syst.*, vol. 26, pp. 47–56, Jan. 2004.

[4] MPI Forum, *MPI: A Message-Passing Interface Standard. Version 2.2*, June 23rd 2009.

[5] T. Hoefler, R. Rabenseifner, H. Ritzdorf, B. R. de Supinski, R. Thakur, and J. L. Traeff, "The Scalable Process Topology Interface of MPI 2.2," *Concurrency and Computation: Practice and Experience*, vol. 23, pp. 293–310, Aug. 2010.

[6] T. Hoefler and J. L. Traeff, "Sparse collective operations for MPI," in *Proceedings of the 23rd IEEE International Parallel & Distributed Processing Symposium (IPDPS), HIPS Workshop*, May 2009.

[7] T. Hoefler, F. Lorenzen, and A. Lumsdaine, "Sparse Non-Blocking Collectives in Quantum Mechanical Calculations," in *Recent Advances in Parallel Virtual Machine and Message Passing Interface, 15th European PVM/MPI Users' Group Meeting*, vol. LNCS 5205, pp. 55–63, Springer, Sep. 2008.

[8] J. Pješivac-Grbović, G. Bosilca, G. E. Fagg, T. Angskun, and J. J. Dongarra, "Mpi collective algorithm selection and quadtree encoding," *Parallel Comput.*, vol. 33, pp. 613–623, Sept. 2007.

[9] A. Bar-Noy and S. Kipnis, "Designing broadcasting algorithms in the postal model for message-passing systems," *Math. Syst. Theory*, vol. 27, no. 5, pp. 431–452, 1994.

[10] R. M. Karp, A. Sahay, E. E. Santos, and K. E. Schauser, "Optimal broadcast and summation in the LogP model," in *Proc. of Symposium on Parallel Algorithms and Architectures*, pp. 142–153, 1993.

[11] P. Sanders, J. Speck, and J. L. Träff, "Two-tree algorithms for full bandwidth broadcast, reduction and scan," *Parallel Comput.*, vol. 35, pp. 581–594, December 2009.

[12] "High performance RDMA protocols in HPC," in *Proceedings, 13th European PVM/MPI Users' Group Meeting*, Lecture Notes in Computer Science, (Bonn, Germany), Springer-Verlag, September 2006.

[13] D. J. A. Welsh and M. B. Powell, "An upper bound for the chromatic number of a graph and its application to timetabling problems," *The Computer Journal*, vol. 10, no. 1, pp. 85–86, 1967.

[14] A. Alexandrov, M. F. Ionescu, K. E. Schauser, and C. Scheiman, "LogGP: Incorporating long messages into the LogP model," *J. of Par. and Distr. Comp.*, vol. 44, no. 1, pp. 71–79, 1995.

[15] A. Gara, M. A. Blumrich, D. Chen, G. L.-T. Chiu, M. E. G. P. Coteus, R. A. Haring, P. Heidelberger, D. Hoenicke, G. V. Kopcsay, T. A. Liebsch, M. Ohmacht, B. D. Steinmacher-Burow, T. Takken, and P. Vranas, "Overview of the bluegene/l system architecture," *IBM Journal of Research and Development*, vol. 49, no. 2, pp. 195–213, 2005.

[16] T. Hoefler, A. Lumsdaine, and W. Rehm, "Implementation and performance analysis of nonblocking collective operations for mpi," in *Proc. of the 2007 ACM/IEEE conference on Supercomputing (CDROM)*, 2007.

[17] M. ten Bruggencate and D. Roweth, "Dmapp - an api for one-sided program models on baker systems," in *Cray User Group Conference*, CUG, 2010.

[18] R. Alverson, D. Roweth, and L. Kaplan, "The gemini system interconnect," in *Proceedings of the 2010 18th IEEE Symposium on High Performance Interconnects*, HOTI '10, (Washington, DC, USA), pp. 83–87, IEEE Computer Society, 2010.

[19] Hyper Transport Consortium, *HyperTransport I/O Technology Overview An Optimized, Low-latency Board-level Architecture*, 2004.

[20] M. Woodacre, D. Robb, D. Roe, and K. Feind, "The sgi® altixtm 3000 global shared-memory architecture," 2005.

[21] T. Hoefler, C. Siebert, and A. Lumsdaine, "Group Operation Assembly Language - A Flexible Way to Express Collective Communication," in *Intl. Conf. on Par. Proc.*, Sep. 2009.

[22] P. Erdős and A. Rényi, "On the evolution of random graphs," in *Publication of the Mathematical Institute of the Hungarian Academy of Sciences*, pp. 17–61, 1960.

[23] W. C. Skamarock and J. B. Klemp, "A time-split nonhydrostatic atmospheric model for weather research and forecasting applications," *J. Comput. Phys.*, vol. 227, pp. 3465–3485, Mar. 2008.

[24] R. Mei, W. Shyy, D. Yu, and L.-S. Luo, "Lattice boltzmann method for 3-d flows with curved boundary," *J. Comput. Phys.*, vol. 161, pp. 680–699, July 2000.

[25] C. Bernard, M. C. Ogilvie, T. A. DeGrand, C. E. DeTar, S. A. Gottlieb, A. Krasnitz, R. Sugar, and D. Toussaint, "Studying Quarks and Gluons On Mimd Parallel Computers," *International Journal of High Performance Computing Applications*, vol. 5, no. 4, pp. 61–70, 1991.

[26] T. A. Davis, "University of Florida Sparse Matrix Collection," *NA Digest*, vol. 92, 1994.

[27] K. Schloegel, G. Karypis, and V. Kumar, "Parallel static and dynamic multi-constraint graph partitioning," *Concurrency and Computation: Practice and Experience*, vol. 14, no. 3, pp. 219–240, 2002.

[28] G. Haase, M. Kuhn, and S. Reitzinger, "Parallel algebraic multigrid methods on distributed memory computers," *SIAM J. Sci. Comput.*, vol. 24, pp. 410–427, Feb. 2002.

[29] J. Bruck, C. T. Ho, S. Kipnis, and D. Weathersby, "Efficient algorithms for all-to-all communications in multi-port message-passing systems," in *6th ACM Symp. on Par. Alg. and Arch.*, pp. 298–309, 1994.

[30] J. S. Vetter and F. Mueller, "Communication characteristics of large-scale scientific applications for contemporary cluster architectures," *J. Parallel Distrib. Comput.*, vol. 63, pp. 853–865, Sept. 2003.

[31] S. Kamil, L. Oliker, A. Pinar, and J. Shalf, "Communication requirements and interconnect optimization for high-end scientific applications," *IEEE Trans. Parallel Distrib. Syst.*, vol. 21, no. 2, pp. 188–202, 2010.

[32] R. Bordawekar, A. Choudhary, and J. Ramanujam, "Automatic optimization of communication in compiling out-of-core stencil codes," in *Proceedings of the 10th international conference on Supercomputing*, ICS '96, (New York, NY, USA), pp. 366–373, ACM, 1996.

[33] L. Renganarayana, M. Harthikote-Matha, R. Dewri, and S. Rajopadhye, "Towards optimal multi-level tiling for stencil computations," in *Parallel and Distributed Processing Symposium, 2007. IPDPS 2007. IEEE International*, pp. 1 –10, march 2007.

[34] S. Krishnamoorthy, M. Baskaran, U. Bondhugula, J. Ramanujam, A. Rountev, and P. Sadayappan, "Effective automatic parallelization of stencil computations," in *Proceedings of the 2007 ACM SIGPLAN conference on Programming language design and implementation*, PLDI '07, (New York, NY, USA), pp. 235–244, ACM, 2007.

[35] S. Potluri, P. Lai, K. Tomko, Y. Cui, M. Tatineni, K. Schulz, W. Barth, A. Majumdar, and D. Panda, "Optimizing a Stencil-Based Application for Earthquake Modeling on Modern InfiniBand Clusters," tech. rep., Ohio State University, 2009. OSU-CISRC-12/09.

[36] E. Gabriel, S. Feki, K. Benkert, and M. M. Resch, "Towards performance portability through runtime adaptation for high-performance computing applications," *Concurr. Comput. : Pract. Exper.*, vol. 22, pp. 2230–2246, Nov. 2010.

[37] S. Kumar, P. Heidelberger, D. Chen, and M. Hines, "Optimization of applications with non-blocking neighborhood collectives via multisends on the blue gene/p supercomputer," in *Parallel Distributed Processing (IPDPS), 2010 IEEE International Symposium on*, pp. 1 –11, april 2010.

[38] R. Das, M. Uysal, J. Saltz, and Y.-S. Hwang, "Communication optimizations for irregular scientific computations on distributed memory architectures," *J. Parallel Distrib. Comput.*, vol. 22, pp. 462–478, Sept. 1994.

[39] A. Faraj, X. Yuan, and D. Lowenthal, "Star-mpi: self tuned adaptive routines for mpi collective operations," in *Proceedings of the 20th annual international conference on Supercomputing*, ICS '06, (New York, NY, USA), pp. 199–208, ACM, 2006.